Measuring Visual Consistency in 3D Rendering Systems

Alfredo Nantes, Ross Brown and Frederic Maire
Queensland University of Technology
Faculty of Science and Technology
Brisbane, Queensland, Australia

a.nantes@qut.edu.au, r.brown@qut.edu.au, f.maire@qut.edu.au

Abstract

One of the major challenges facing a present day game development company is the removal of bugs from such complex virtual environments. This work presents an approach for measuring the correctness of synthetic scenes generated by a rendering system of a 3D application, such as a computer game.

Our approach builds a database of labelled point clouds representing the spatiotemporal colour distribution for the objects present in a sequence of bug-free frames. This is done by converting the position that the pixels take over time into the 3D equivalent points with associated colours. Once the space of labelled points is built, each new image produced from the same game by any rendering system can be analysed by measuring its visual inconsistency in terms of distance from the database. Objects within the scene can be relocated (manually or by the application engine); yet the algorithm is able to perform the image analysis in terms of the 3D structure and colour distribution of samples on the surface of the object.

We applied our framework to the publicly available game *RacingGame* developed for Microsoft® Xna®. Preliminary results show how this approach can be used to detect a variety of visual artifacts generated by the rendering system in a professional quality game engine.

Keywords: Synthetic Image Analysis, Computer Vision, Computer Game Testing.

1 Introduction

One of the major challenges facing a present day game development company is the removal of bugs from such complex synthetic environments. Today, games provide remarkably realistic graphics with highly interactive scenarios held together by complex software that requires large development and testing teams. The more complex the software, the more crucial the effort of the companies in testing their product in order to ensure a high enough quality in terms of functionality, stability and robustness in general. Furthermore, a computer game is not only expected to work properly but it has to be, amongst other things, fun, challenging, realistic and well animated.

As observed by Macleod (2005), game play typically consists of a set of actions that move the game through a number of successive states. Testing has the objective of ensuring that there is no combination of actions that brings the game to a state such that the experience meant to be offered gets corrupted. In recent years, researchers have been investigating the problem of defining and measuring the game play experience through cognitive models (Ermi & Mäyrä, 2005; C. A. Lindley & Sennersten, 2006; C. A. Lindley & Sennersten, 2007) or qualitative features (Kapoor, Burleson, & Picard, 2007; Roberts, Strong, & Isbell, 2007; Yannakakis & Hallam, 2007). These works have mainly attempted to model playability issues such as gameplay functionality, game usability and game mechanics.

In a previous work (Nantes, Brown, & Maire, 2008), we have classified these issues as the *Entertainment Inspection* component of game testing, distinguishing them from the complementary *Environment Inspection* issues. We defined the latter to be the degrees to which a game environment is perceived as being consistent in terms of sound, textures, meshes, lights and shadow effects that all contribute to the game play experience. An example of a visually inconsistent game environment is shown in Figure 1.

Focusing on the Environmental Integrity Inspection problem, we introduced an image processing approach to the automatic testing of such problems within 3D virtual environments and games (Nantes et al., 2008). The implementation of such an automated environment debugger can bring important benefits to the game industry. Indeed, it will bring costs savings by decreasing the number of play testers required during the Quality Assurance process. Moreover, it will increase the robustness of the product to release by allowing the coverage of a far larger set of test cases than currently performed by human players.

In this paper we present a complete and generalisable pixel transformation technique for the testing of 3D virtual environments that is feasibly able to be applied to any form of game engine that utilizes present rendering technology. It is generalisable, in being able to handle any scene using modern shader rendering, and enables the geometry and texturing in an environment to be tested automatically.

In Section 2 we review present work in the area of environment testing and geometry reconstruction. Section 3 presents our unique formulation to solve the scene integrity problem. The technique's theoretical framework is presented in Section 4. In Section 5 we present the algorithm we used for measuring the visual consistency between two frames. Section 6 concludes this paper with some remarks on preliminary results and future work.

(a) (b) (c) (d)

Figure 1: Examples of visually inconsistent game environment. The pictures show some examples of mesh and texture corruption. Anomalies such as mesh corruption (present in all pictures) may occlude big areas of the screen, thus compromising the normal user-application interaction. Pictures (c) and (d) depict examples of texture corruption by which textures are wrongly mapped (the leaves of the palm trees in picture c are red; the car and the trees in picture d have transparent textures).

2 Related Work

Most work in automated game assessment is concentrated on the areas of game play via the use of artificial intelligence approaches to human entertainment modelling, for measuring qualitative factors such as challenge and curiosity (Yannakakis & Hallam, 2007) user interaction experiences (Yang, Marsh, & Shahabi, 2005), integrity and fairness in rudimentary computer games (Macleod, 2005), amongst others used to find faults in game mechanics (Chan, Denzinger, Gates, Loose, & Buchanan, 2004).

Although all these results could be used for assisting the design of the AI (Artificial Intelligence) component of a game, they do not address important environment integrity issues that surely affect the playability of the game environment. The small amount of Environmental Integrity Inspection research, has concentrated on algorithms for mesh geometry testing to find holes and slivers, (Cheng, Tamal, Edelsbrunner, Facello, & Teng, 1999) some techniques ending up in commercial systems such as, for example, the Half Life 2 Level Editor Hammer (Valve, 2009).

To our knowledge, no one else has attempted to use the image generated by a rendering system to reverse engineer the geometric contents of a scene, in order to perform geometry and texture integrity testing. In this work we propose a generalised image-space computer vision technique that addresses the automation of the Environment Integrity problem for a large class of geometry and texturing bugs.

Our new technique exploits the ability of the software rendering system to reverse the object transformations used to derive the final synthesized image. This inverse camera transform approach has similarities to computer vision techniques that exploit stereopsis and motion parallax from image data to generate 3D object geometry (Rusu, Blodow, Marton, & Beetz, 2008). These methods seek to generate the camera transform from noisy CCD imagery, to create a mesh representing the objects in the image. In these techniques, the control of accuracy is problematic, due to errors in the system caused by camera noise, sampling position estimation inaccuracies, and antialiasing caused by the limited sampling resolution available in video cameras.

In our technique the problems of correct geometry generation are largely removed, as we already have the transform that has been applied to the pixel in order to colour and position it in screen space. Since transforms are already known, and we have control over the rendering environment, the original object geometry can be constructed from the series of test images presented, giving a superior approach to automated object geometry and texture checking, in the original normalized object coordinate system. We now proceed to describe this new inverse transformation technique in detail.

3 Problem Formulation

A *picture* can be defined by the way an observer perceives the distribution of colours on a specific support such as a piece of paper or a screen (Ning, 1997). The perception of a picture in terms of the meaning it has for the subject necessarily depends – among other things – on the observer experience (Giorgi, 2009).

According to this point of view, the correctness or consistency of a picture inherently depends on the observer perceptive capabilities and experience. Yet, a user-independent analysis can be done under the assumption that a number of visually consistent pictures can be generated – that is to say, images that are perceived as correct by some observer, e.g. the game designer.

By using a description similar to the one introduced by Fink et al. (2007), a game can be thought of as a two-function system. With the passage of time the game takes on a sequence $s_0, s_1, s_2, \ldots (s_i \in S)$ of states. Each state is obtained from the preceding one by the function

$$G_{\text{upd}} : S \times A \rightarrow S \tag{1}$$

where A is the input set of actions that can be performed by the player. The output of the game can be described by another function

$$G_{\text{out}} : S \rightarrow O \tag{2}$$

that produces the frame O from the current state S. Intuitively, the set O is correct if both S and G_{out} are correct.

In this work we assume we know that a collection of outputs in O was correct. We call such a collection the set of *validated frames* and we denote it by F_v. As we cannot say anything about the correctness of G_{out}, any output O not in the set of validates frames can be either correct or wrong. We call this collection the set of *test frames* that we denote by F_t. Note that the intersection of F_v and F_t may not be the empty set as S may contain repetitions.

With this work we aim to infer about the correctness of G_{out} via measuring the visual consistency of test frames

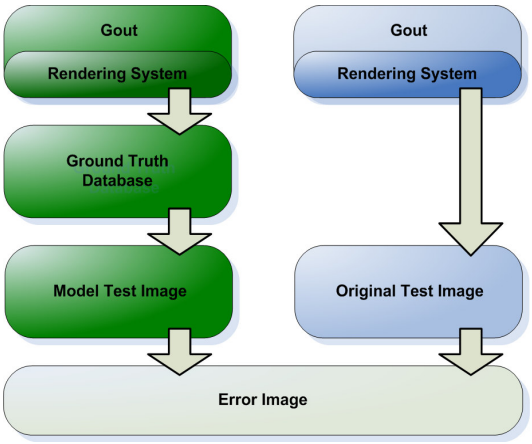


Figure 2: Framework for the detection of visual inconsistencies in test images. The left hand side of the diagram shows the process of building the ground truth database from validated images produced by some rendering system. Images are first converted to spatiotemporal colour distributions that form the ground truth database. Then models of the test images are generated for visual consistency measurements.

given a collection of validated frames and some information about G_{out} and its input S.

In this paper, unless otherwise specified, we will use G_{out} interchangeably to denote the output function and its hardware and/or software implementation.

4 Theoretical Framework

If the test frames were produced from the same game states that produced the validated frames, the visual consistency of each frame in F_t would reduce to a mere pixel-by-pixel comparison with the related frame in F_v. Indeed, since the input states do not change and the output F_v is visually consistent by definition, any new output from G_{out} that differs from the related frame in F_v can be considered visually inconsistent.

Typically, the function G_{out} is partially implemented via software and partially via hardware. The hardware part corresponds to the GPU of the machine in which the game runs. Because a computer game is typically expected to run on a number of different GPUs, there will be a number of different implementations for some G_{out}. Therefore, it may well be that the same input state results in a different output frame for some G_{out}.

Instead of addressing the problem of measuring the correctness of G_{out} through a solution for reproducing a specific sequence of input states, we build a 3D model for each object that will be rendered in the collection of validated images F_v assuming that we know the set of geometric transformations that will be applied to the model by the rendering system. In addition to the RGB colour of the model in the frame, our validated models contain further information about the colour distribution by which they appear in F_v. Since the validated frames are correct, these models will represent the ground truth database for all objects appearing in a new frame. To put it another way, each model we generate is a spatiotemporal colour distribution that enables us – with the due assumptions – to model even the appearance of animated or deformable objects, without knowing anything about the actual structure or dynamics of their components. Figure 2 depicts the overall process of error detection.

4.1 Background

Any output frame is synthesized by G_{out} through a rendering process that takes objects in their original object or local space and turns them – via some geometric transformations – into the screen space to form the scene we perceive. Such a process is shown in Figure 3 wherein only the space transformation operations are depicted. Readers who are interested in a more comprehensive description of the rendering process can refer to Luna (2006).

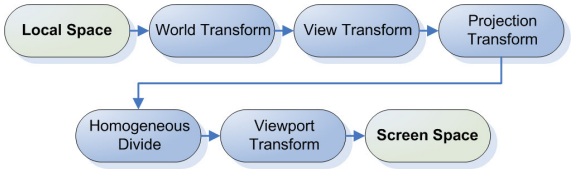


Figure 3: A GPU pipeline showing the process involved in transforming coordinates from local to screen space.

The *World* and *View* stages in the picture are the affine transformations that give the position in eye space. To represent affine transformation with matrices a homogeneous component w is introduced so that any vector (x, y, z) becomes (x, y, z, w) where w is typically set to 1. The *Projection* transformation then gives a position in a coordinate system bounded by the homogeneous unit cube: the *clip* space. Next, the position in normalized device coordinates is given by the *Homogeneous Divide*. This step brings the homogeneous component back to 1 after carrying out the projection matrix multiplication. Finally, the *Viewport* transformation stretches and translates the projected coordinates to fit a specific position of the screen.

The final colour of a *fragment* – a portion of the final image of the size of a pixel – is decided by the lighting and shadowing techniques implemented by the designer as part of G_{out}.

The rendering process of Figure 3 is reversible in that, fragments of the screen can be brought back to their original position in object space. Let $\mathbf{S}$ be the position of a fragment in screen space and M the world-view-projection matrix that transforms the object to which the fragment belongs from original position to screen space position. The following relations hold:

$$\mathbf{D} = \mathbf{S} \times M_v^{-1} \times M^{-1} \tag{3}$$

$$\mathbf{R} = \frac{\mathbf{D}}{\mathbf{w}} \tag{4}$$

where M_v is the *viewport* matrix of the frame; $\mathbf{R}$ is the original position of the fragment equivalent to $\mathbf{S}$ given M and M_v and $\mathbf{w}$ is the homogeneous coordinate component of $\mathbf{D}$. We denote a fragment in its original

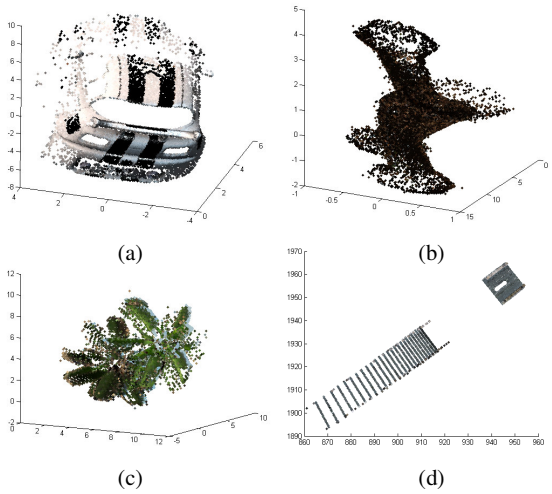


Figure 4: Examples of vectors (**R**,**C**) for four different objects observed in a collection of frames, namely a car (a), the trunk of a palm tree (b), the leaves of a palm tree (c) and a segment of the track (d).

position in object space by the vector $(\mathbf{R},\mathbf{C})$ where $\mathbf{C}$ is the final colour that the fragment takes at the end of the rendering process. Note that the perspective-projection matrix is, in general, singular due to the lost of the z dimension; individual points in eye space that lie along the same line of projection will project to a single point. However, since rendering systems map the coordinate values within the view volume to a (unit) cube, the information about z is kept and the matrix becomes invertible (Van Verth & Bishop, 2004). Finally, note from Figure 3 that the *rescaling* operation performed by Equation 4 should take place after inverting the Projection transform and before multiplying by the inverse of the World-View matrix. However, because no affine transformation (World and View) can alter the homogeneous component of a coordinate vector, this operation can be performed at the end of the reversing process. The advantage of postponing the rescaling operation is to compute the inverse of two matrices (the Viewport and the combined World-View-Projection) instead of three (Viewport, Projection and the combined World-View).

4.2 Visual Consistency Error in Object Space

By applying Equation 3 and 4 to all fragments of an object in a frame we end up with a vector of 3D coloured points that model the spatial and colour distribution of those parts of the object visible in that frame. The colours

Figure 5: Example of *object map* (left) and *matrix map* (right) of a frame. Colours identify different objects in the object map and different world-view-projection matrices in the matrix map. The depth buffer is encoded by the α channel of the object map.

are taken directly from the final image, the *frame buffer*. As long as $\mathbf{S}$, M_v and M are known $(\mathbf{R},\mathbf{C})$ can be built from an arbitrary number of frames. Figure 4 depicts four examples of the vector $(\mathbf{R},\mathbf{C})$ for different objects that have been built from a number of frames. Such objects form the ground truth database of Figure 2 by which the test images will be assessed.

Definition 1: Visual Consistency of an Object in Object Space

Given a set A of fragments of an object in object space relative to a single frame and the set B of fragments of the same object in object space relative to some collection of frames; A is ε *consistent* with B in object space if

$$d_o(A,B) \le \varepsilon \tag{5}$$

We call $d_o(A,B)$ the *visual consistency error* of A relative to B in object space.

Henceforth, ε is assumed to be a small constant.

Definition 2: Visual Consistency of a frame in Object space

A frame F_t is ε *consistent* with a collection of frames F_v in object space if the set A is ε *consistent* in object space with the set B for all objects in F_t.

4.3 Building the vector $(\mathbf{R},\mathbf{C})$ through the game engine

From Definition 2 it follows that in order to decide whether a test frame is correct or not we need to compute the vectors $(\mathbf{R},\mathbf{C})$ for all objects appearing in that frame and in some given collection of frames. Such process, entails $\mathbf{S}$ and M to be known for each object, for each frame. Without loss of generality, we can assume that M_v is constant for all frames generated by G_{out}.

During the rendering process a graphics application (such as a computer game) passes the objects that need to be rendered to the GPU as well as – among other things – their geometric transformations. The screen space coordinates of the fragments are then computed by the GPU. Therefore, to determine both $\mathbf{S}$ and M for each object we need the GPU to label with $f(O)$ those fragments of the current frame belonging to the object O and with $g(M)$ those fragments generated from the transformation M. The functions f and g should produce a unique colour identifier given an object or a matrix as input. By doing so we effectively generate two images that allow us to map the colours of $\mathbf{S}$ and M to the related object O and matrix M. We call the image generated through f the *object map* and the image generated through g the *matrix map*.

To make this process as little game software dependent as possible we created a class *LabelMapShader* that inherits from the same class used to initialize any other shader effect in the game. As with the other render classes of the game, our class is instantiated once and used whenever an object needs to be rendered. The label map class functionality is turned on and off by

Listing 1. Pseudocode of our *LabelMapShader* class.

Attributes:
OM, *MM*: TEXTURE
objColour,*mtxColour*: EFFECT_PARAMETER
objColTab: TABLE<String, Vector>
mtxColTab: TABLE<Matrix, Vector>

Methods:
updateMaps(*O*, *M*)
set *objColour* to *f*(*O*)
if (*O.name* not in *objColTab*)
add to <*O.name*, *objColour*> to *objColTab*
end
set *mtxColour* to *g*(*M*)
if (*M* not in *mtxColTab*)
add <*M*, *mtxColour*> to *mtxColTab*
end
set render targets to *OM* and *MM*
commit changes to the GPU

generateMaps(*O*)
set *LabelMapping* to be the current shader technique
render *O*

debug flags in the game source code. The pseudocode of such a class is shown in Listing 1. Object and matrix maps – identified by the attributes *OM* and *MM* respectively – are implicitly updated by the function *generateMaps*(). When an object needs to be rendered the game first calls the function *updateMaps*() which adds the object name and matrix along with the associated colours to the tables *objColTab* and *mtxColTab*. In this way, colours are linked to the related objects in the object map and to the related matrices in the matrix map. Then the *updateMaps*() function sets the colours to be used by the shader for rendering the object *O* to the two textures *OM* an *MM*. The function *generateMaps*() – which is called by the game right after the function *updateMaps*() – sets the shader technique to use whose code is shown in Listing 2. Then it sends the object *O* to the GPU to have it rendered on the texture *OM* with the colour *objColour* and to the texture *MM* with the colour *mtxColour*.

Listing 2. Shader Code that generates the object and matrix maps for the current frame.

```
VSOutput OMMap_VS(VSInput In)
{
   VSOutput outVS = (VSOutput)0;
   outVS.Pos = mul(float4(In.Pos.xyz, 1.0f),
                                   worldViewProj);
   outVS.Depth = float2(outVS.Pos.z,
                                   outVS.Pos.w);
   return outVS;
}

PSOutput OMMap_PS(VSOutput In)
{
   PSOutput outPS = (PSOutput)0;
   outPS.ObjCol.xyz = objColour.xyz;
   outPS.ObjCol.w = (In.Depth.x)/(In.Depth.y);
   outPS.MtxCol = mtxColour;

   return outPS;
}
```

The function f we used simply returns a four-byte identifier set to the current value of an internal counter. The counter is incremented only if the name of the next object to render is not already in the table *objColTab*. Bytes are then treated as (R,G,B,α) colour components. Of this vector, only the (R,G,B) components are used for colouring the pixels. The α channel is replaced by the value of the depth buffer which is computed by the same shader code used to generate the maps. This is shown in Listing 2. As it can be noted, the pixel shader renders to two targets, namely *OM* and *MM* corresponding to the object map and matrix map texture respectively. The variable *objColour* containing the (R,G,B) components of the colour to be rendered is stored in the first three channels of the target *OM*. The forth channel of *OM* is set to the normalized z value of the screen space position computed by the vertex shader, namely the depth of the pixel. Therefore, the object map texture completely defines $\mathbf{S}$ through the table *objColTab*. Such a texture represents the final image segmented at the object level. The colour of the segments is specified in *objColTab*.

For the matrix map, fragments of the object that have undergone the transformation M take the colour generated through the function g and stored in the table *mtxColTab*. The colour returned by g is a four-byte array set to the current value of an internal counter. This counter is incremented only if the matrix of the next object to render is not already in the table *mtxColTab*. This time all (R,G,B,α) components of the array are used for colouring the pixels. Figure 5 depicts an example of such maps related to a frame. Note that, with this implementation, the object map can map up to 2^{3d} objects per frame where d is the colour depth used. For a 32-bit *truecolor* image (8 bits per channel) the maximum number of objects that can be stored for a single frame is $\approx 1.6 \cdot 10^7$. Likewise, the maximum number of transformation matrices that can be mapped by the matrix map is 2^{4d} that is $\approx 4.3 \cdot 10^9$ matrices for a *truecolour* image.

4.4 Visual Consistency Error in Screen Space

The process described in the previous Section can be used for computing the vector $(\mathbf{R},\mathbf{C})$ of an object in object space, given a frame or a set of validated frames in which the object appears. This requires the normal rendering process of G_{out} to be modified. Indeed, two additional steps need to be performed for each object to be rendered namely, the building of the object and matrix maps.

If the rendering pipeline cannot be modified during testing the visual consistency can still be measured in screen space. To that end, the vector $(\mathbf{R},\mathbf{C})$ needs to be converted to the equivalent vector $(\mathbf{S},\mathbf{C})$ where $\mathbf{S}$ is the screen space position equivalent to $\mathbf{R}$. This can be done by using the same geometric transformations used by the

game to render the test frame. Since $(\mathbf{R},\mathbf{C})$ models the objects rendered by the game, this process will create a model of the test frame. For this reason, we shall call the vector $(\mathbf{S},\mathbf{C})$ the *model test frame*.

This transformation process, shown in Listing 3, consists of an emulation of the rendering process in the graphics pipeline and assumes that some information about the test image is available. Apart from the vector $(\mathbf{R},\mathbf{C})$, also the transformation matrix M, the viewport matrix M_v and the depth buffer Z of the test frame need to be known *a priori*. Note that M, M_v and Z can be extracted from the input and output of the rendering pipeline without modifying the rendering process itself.

After computing the normalized device coordinates through the function *normalize*(), points outside the normalized volume of space (*frustum*) are clipped through the binary mask generated by the function *xyzClipMask*(). To be sure, also the colour information about the points needs to be updated. This is done by masking the colour vector with the same *cMask* used for clipping the device coordinates.

Once the device coordinates have been computed the GPU performs the *backface culling* (Luna, 2006). However, as our object model $(\mathbf{R},\mathbf{C})$ is made of points and not polygons, there are no faces to remove but sets of points that model them. Assuming that we have the depth buffer Z of the test frame we can use it as *z-test* function. That is, we remove from the vector *scrCoords* those points whose z value does not match with the value of the depth buffer at screen position (*scrCoords.x, scrCoords.y*). The variable *zMask* is then a binary vector whose elements are 1 if the relation

$$Z(scrCoords.x, scrCoords.y) = scrCoords.z \qquad (6)$$

holds; 0 otherwise.

When computing $\mathbf{R}$ in Equation 4 the approximated version of $\mathbf{S}$ is used as the screen space coordinates are read from the image and the depth buffer, both of finite resolution. This produces the aliasing effect of scattered points visible in Figure 4. Hence, because of the discretization error introduced when building $(\mathbf{R},\mathbf{C})$, Equation 6 needs to be modified so as not to clip too many valid points from the *scrCoords* vector. That is, we need to allow for some tolerance τ when performing the z test. Because of the non-linearity of the depth buffer this tolerance will be a function *zTest*() of τ and the depth of each pixel in the image which will be read from the depth buffer Z. The result of this last operation is the screen space vector $(\mathbf{S},\mathbf{C})$ equivalent to $(\mathbf{R},\mathbf{C})$.

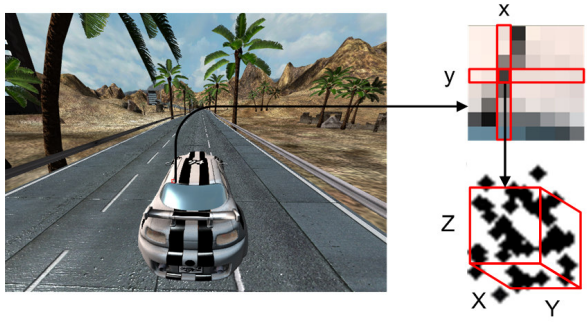


Figure 6: Relationship between a fragment in screen space and the related region of the vector (**R**,**C**). The cube in object space is the equivalent region of the fragment at screen position (*x*,*y*). The number of points *p* in the cube corresponds to the number of points in the model test frame at (*x*,*y*).

Listing 3. Pseudocode for converting the vector (**R**,**C**) to the vector (**S**,**C**).

```
function transformRC((R,C), M, Mv, Z, τ) returns (S,C)

    devCoords   = normalize(R*M)
    cMask       = xyzClipMask(devCoords)
    devCoords   = devCoords(cMask)
    colorSet    = C(cMask)
    scrCoords   = devCoords*Mv
    zMask       = zTest(scrCoords, Z, τ)
    scrCoords   = scrCoords(zMask)
    colorSet    = colorSet(zMask)

    return (scrCoords,colorSet)
```

It is important to note that the model test frame may not be the exact copy of the test frame. More precisely, a fragment of the test frame at screen position (x, y) coincides with a number p of fragments of the model at the same position. The number p depends on the density of the vector $(\mathbf{R},\mathbf{C})$ in the object space region equivalent to the pixel (x, y). Such a density, in turn, depends on the distances z in screen space at which the same region of $(\mathbf{R},\mathbf{C})$ has been observed in the validated frames; the bigger such a distance, the sparser the region.

The relation between p and the density of $(\mathbf{R},\mathbf{C})$ is illustrated in Figure 6. If p is zero, the related pixel of the model test frame will not contain fragments from $(\mathbf{R},\mathbf{C})$. Empty regions of the model test frame will be further discussed in Section 5.

Finally, it should be noted that the colour of the p fragments at (x, y) may not be the same as the colour of the pixel at the same position. This is due to the different light conditions under which the same region of $(\mathbf{R},\mathbf{C})$ has been observed in the validated images.

Once we have the vector $(\mathbf{S},\mathbf{C})$ we can measure the visual consistency in screen space.

Definition 3: Visual Consistency of an Object in Screen Space

Given a set A of fragments of an object in screen space relative to a single frame and the set B of fragments of the same object in screen space relative to some collection of frames; A is ε *consistent* with B in screen space if

$$d_s(A,B) \le \varepsilon \qquad (7)$$

We call $d_s(A,B)$ the *visual consistency error* of A relative to B in screen space.

Listing 4. Pseudocode for measuring the visual consistency of a test frame.

```
function measureVC(A, B) returns two images

    local variables:  qThres, quadTree threshold
                      dThres, node density threshold
                      r, patch radius

  frBuffer = frame(A)
  setQ = qTreePartition(frBuffer, qThres)
  for each node n in setQ
     colVectN = colour(A,n)
     mColourN = mean(colVectN)
     if density(B,n) < dThres
        nCentre = centroid(n)
        build a patch P or radius r centered at nCentre
        setP = points of B.S in P
        if setP is empty
           vConst(n) = ∞
           spDist(n) = ∞
           continue
        end
        setO = kNN of nCentre among the points of setP
        distO = mean of distances from nCentre to setO
     else
        setO = points of B.S in n
        distO = 0
     end
     colVect = colour(B,setO)
     setC = kNN of mColourN among the colours of
                                                    colVect
     mu = mean(setC)
     sigma = covariance(setC)
     vConst(n) = mgd(nColour, mu, sigma)
     spDist(n) = distO
  end
  return vConst, spDist
```

Definition 4: Visual Consistency of a Frame in Screen Space

A frame F_t is ε *consistent* with a collection of frames F_v in screen space if the set A is ε *consistent* in screen space with the set B for all objects in F_t.

5 Visual Consistency Measurements

This section presents an algorithm for measuring the visual consistency error in screen space. Before reviewing the algorithm, however, it is important to clarify the concept of density of the model test image and explain why it is factored into the error measurement process.

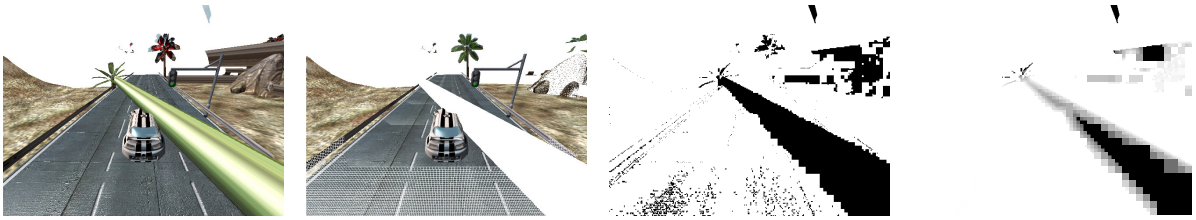
Consider the model test frame in Figure 7b. By comparing it with the original test frame, it can be noticed that the model has sparse or even empty regions. As argued in Section 4.4, sparse or empty regions in $(\mathbf{S},\mathbf{C})$ may be due to equivalent sparse or empty regions in the vector $(\mathbf{R},\mathbf{C})$. However, an empty region in the model can also be caused by an object that does not have a related $(\mathbf{R},\mathbf{C})$ because it is not present in the collection of validated frames. Unfortunately, if an object is not present in such a collection we cannot claim that the related empty region is an anomaly as we are not assuming that the validated frames contain all possible objects of the game. Nor should we claim that the region is correct as the information we need for measuring the visual consistency is missing. We can then say that empty regions in the model test image are areas of high uncertainty with respect to the visual consistency error that can be measured on them. Conversely, a dense region of the model is a region that has been observed from the same or a closer distance in the collection of validated frames and/or from different camera angles. Hence, the measure of its visual consistency is expected to be accurate. Dense regions in the model test frame are then areas of high certainty. The sparseness of a region in the model is therefore a measure of confidence with which we should accept the visual consistency error computed on the same regions.

In this work we used the Multivariate Gaussian Distribution (MGD) to measure the visual consistency error in screen space. However, instead of computing the distance for each objects in the test frame, we computed the distance for rectangular supports of a quadtree subdivision. Listing 4 shows the pseudocode of our implementation.

After extracting the test frame from A through the function *frame*() a quadtree partitioning is applied to it. A quadtree node (quad) is no longer sub-divided if the difference between the maximum and the minimum values of each colour component in the node is smaller than its respective threshold *qTrhes*. In our experiments such a threshold was set to 0.5 for all colour components. We found this being a good trade-off between speed and accuracy. The output of the partitioning is the set *setQ* of nodes that make up the quadtree. Original and model test frame share the same partitioning.

The node density – computed by the function *density*() – is a measure of the sparseness of the model B for the current node. However, because nodes have been partitioned on the basis of their colour homogeneity and not their sparseness, small nodes in sparse (non-empty) regions may have zero density, which would be interpreted as maximum uncertainty. To allow for this, we build a patch of radius r that surrounds the low density node. If the patch still does not contain $(\mathbf{S},\mathbf{C})$ points the node is considered empty and it will be assigned the maximum distance and uncertainty. Otherwise, to find the most appropriate surrounding $(\mathbf{S},\mathbf{C})$ points by which to measure the consistency error we use the k-Nearest Neighbours (kNN) algorithm. Specifically, we compute the k nearest model points in the neighbourhood of the node centre were k is the minimum value between 10 and the set of model points in the patch. By storing the mean radius of such a set, the variable *distO* represents the confidence in making any inference about the node. If the node is dense, there are enough $(\mathbf{S},\mathbf{C})$ points to compute the visual consistency error and to expect it to be accurate ($distO = 0$).

Finally, the error is computed with the k colours of the model test frame closest to the mean colour of the node.



(a) (b) (c) (d)

Figure 7: Example of original test frame (a), model test frame (b), visual consistency map (c) and confidence map (d). Sparse regions in the model test frame correspond to equivalent regions of low density in the vector (**R**,**C**). Black regions in the Visual Consistency Map correspond to high inconsistencies with respect to the vector (**S**,**C**). Dark regions in the Confidence Map correspond to areas of low confidence about the consistency measure.

The parameter k is set to be the minimum value between 10 and the number of colours in *ColVect*. From the set of colours extracted, the mean and variance are computed. The consistency error is returned in terms of likelihood – through the multivariate probability density function – of the mean colour of the node given the spatially close fragments observed in the collection of validated frames. Low probabilities correspond to high inconsistencies.

The algorithm generates two images namely, *vConst* encoding the visual consistency error for all nodes of the partitioned frame; and *spDist* representing the confidence about the consistency measured for each node. We call the image *vConst* the *Visual Consistency Map* and the image *spDist* the *Confidence Map*.

Figures 7c and 7d show the output of the algorithm for the test image in Figure 7a. In order to make the *vConst* image visually interesting we only displayed those nodes whose consistency error was below a certain threshold ($1 \cdot 10^{-6}$ in this case). Areas of high uncertainty (dark regions) in the Confidence Map are located in correspondence of empty or sparse regions of the model test frame. The original test frame in Figure 7 is an example of a buggy test image affected by texture corruption (the texture of the palm tree leaves in the background is wrong) and polygon corruption (polygons from the small palm tree and the rocks in the background are wrongly projected). As can be observed from the output of our algorithm such artifacts are correctly detected.

6 Conclusions

This work introduces a general approach for measuring visual inconsistencies in synthetic images generated by a graphics application, such as a computer game. The approach is independent from the 3D application, as it mainly relies on the standard input and output data of the rendering system, and does not require a large amount of source code modification to be implemented within a typical game engine.

Although the accuracy of our implementation is yet to be measured, preliminary results show that the framework is able to cope with geometry and texturing anomalies such as polygon and texture corruption without requiring any knowledge about the geometric primitives and the textures used by the application. It is postulated that the algorithm presented in Listing 4 will perform well for those environments where the colour and geometry of a single object does not (sensibly) depend on the rest of the scene. Colour changes due to *global effects* such as shadows and light reflections and refractions are unlikely to be effectively captured or measured by our algorithm. If the colour or the geometry of fragments sensibly varies over time a more complex mechanism is required. Such a mechanism should be able to elicit possible causal relationships out of specific visual events in the set of validated images and allow for those causes to make inference about similar events in new images.

Our framework has been applied to the publicly available game *RacingGame* a close to professional quality game engine that has been developed as *starter kit* for Microsoft® Xna®[1]. Such a game turned out to be a good test bench for our framework for the following reasons:

- due to its prototypal nature, all visual inconsistencies that we have observed were generated by the game and never induced by us;
- the world position of some objects in the environment (like buildings, banners and trees) was automatically changed by the game at each play session. This allowed us to test our framework for volatile virtual environments;
- render effects are such that the colour and geometry of the objects does not sensibly depend on the rest of the scene.

As far as concerns the game we have tested, the consistent appearance of the objects was correctly discriminated from other visual inconsistencies.

Our experiments show that the Multivariate Gaussian Distribution gives a good measure of the visual inconsistency error for high density regions of the model test frame. However, it can be noticed that sparse regions in the model are likely to be classified as bugs regardless the actual consistency of the equivalent areas in the original test frame.

In our future work we will investigate the use of other functions for measuring the inconsistency error such as the *Kullback-Leibler* divergence and the *Hausdorff* distance. Also, we mean to effectively combine the Visual Consistency Map with the Confidence Map in order to improve the robustness of the solution. To this end, we will build a Bayesian detector from these two maps. After segmenting the visual consistency map into inconsistent regions, we will weigh the inconsistent regions with respect to the confidence map. Those

[1] http://creators.xna.com/en-US/education/starterkits/

regions whose total weights are above a threshold will be flagged as likely to be bugs.

Finally, we mean to provide a consistent way of measuring the accuracy of our algorithm for a proper validation via testing with a larger set of varying scenes, first evaluated for errors by subjective viewers.

7 References

Chan, B., Denzinger, J., Gates, D., Loose, K., & Buchanan, J. (2004). *Evolutionary behavior testing of commercial computer games*. Paper presented at the 2004 IEEE Congress on Evolutionary Computation.

Cheng, S., Tamal, K. D., Edelsbrunner, H., Facello, M. A., & Teng, S. (1999). Sliver Exudation. *Journal of the ACM, 47*(5), 883 - 904.

Ermi, L., & Mäyrä, F. (2005). *Fundamental Components of the Gameplay Experience: Analysing Immersion.* Paper presented at the Digital Games Research Association (DiGRA-05), Perth, Australia.

Fink, A., Denzinger, J., & Aycock, J. (2007). *Extracting NPC behavior from computer games using computer vision and machine learning techniques.* Paper presented at the IEEE Symposium on Computational Intelligence and Games, Hawaii.

Giorgi, A. (2009). The Phenomenological Mind: An Introduction to Philosophy of Mind and Cognitive Science. *Journal of Phenomenological Psychology, 40*(1), 107-108.

Kapoor, A., Burleson, W., & Picard, R. W. (2007). Automatic prediction of frustration. *International journal of human-computer studies, 65*(8), 724-736.

Lindley, C. A., & Sennersten, C. C. (2006). *A Cognitive Framework for the Analysis of Game Play: Tasks, Schemas and Attention Theory*. Paper presented at the 28th Annual Conference of the Cognitive Science Society.

Lindley, C. A., & Sennersten, C. C. (2007). Game Play Schemas: From Player Analysis to Adaptive Game Mechanics. *International Journal of Computer Games Technology, 2008*(2).

Luna, F. (2006). *Introduction to 3D Game Programming with Direct X 9.0c: A Shader Approach (Wordware Game and Graphics Library)*. Plano, Texas: Wordware Publishing Inc.

Macleod, A. (2005). *Game design through self-play experiments*. Paper presented at the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology.

Nantes, A., Brown, R., & Maire, F. (2008, 22-24 October). *A Framework for the Semi-Automatic Testing of Video Games.* Paper presented at the Artificial Intelligence and Interactive Digital Entertainment (AIIDE-08), Stanford University, Palo Alto, California.

Ning, L. (1997). *Fractal Imaging*. San Francisco, California: Morgan Kaufmann Publishers Inc.

Roberts, D., Strong, C., & Isbell, C. (2007). *Estimating Player Satisfaction Through the Author's Eyes.* Paper presented at the Workshop on Optimizing Player Satisfaction Artificial Intelligence and Interactive Digital Entertainment (AIIDE-07), Stanford University, Palo Alto, California.

Rusu, R. B., Blodow, N., Marton, Z. C., & Beetz, M. (2008). *Aligning Point Cloud Views using Persistent Feature Histograms.* Paper presented at the 21st IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Nice, France.

Valve. (2009). Valve Developer Community. Retrieved June 2nd, 2009, from http://developer.valvesoftware.com/wiki/Main_Page

Van Verth, J., & Bishop, L. (2004). *Essential Mathematics for Games and Interactive Applications: A Programmer's Guide*. San Francisco, California: Morgan Kaufmann Publishers Inc.

Yang, K., Marsh, T., & Shahabi, C. (2005). *Continuous archival and analysis of user data in virtual and immersive game environments*. Paper presented at the ACM workshop on Continuous archival and retrieval of personal experiences.

Yannakakis, G. N., & Hallam, J. (2007). Towards Optimizing Entertainment in Computer Games. *Applied Artificial Intelligence, 21*(10), 933-971.